""" src2doc.py – Takes a source file and makes a document file. Default the source is expected to be Python, and the generated document will be LaTeX.

 """

```
rcsFile = "$RCSfile: src2doc.py,v $"
rcsDate = "$Date: 2001-08-09 12:33:27 $"
rcsRev = "$Revision: 1.2 $"
rcsState = "$State: Exp $"
rcsAuthor = "$Author: aa $"
rcsLog = """
$Log: src2doc.py,v $
Revision 1.2  2001-08-09 12:33:27  aa
Tupple stuff (changes in Python between 1.5.2 and 2.0.1?)

Revision 1.1  2000/07/25 13:42:14  aa
Moved to notebook (aa)

Revision 0.4  1997/11/29 18:45:56  anders
Added support for raw string format (r|R) in Python 1.5.

Revision 0.3  1997/10/17 22:10:13  anders
Changed the doc string cleaned up the layout for the Src and Doc class.

Revision 0.2  1997/07/10 13:35:45  anders
Made all values in option dictionaries strings.

Revision 0.1  1997/05/26 00:01:37  anders
Initial working version based on src2ltx v0.1.
"""
```

""" We are importing some library modules. We need `regex` because we are using regular expression to describe source tokens. The `string` module gives us functions for string manipulation. We need the stdio stuff from `sys` and we use `types` to know the types of some objects. """

```
import regex                    # Use the new "re" next time
import string
import sys
from types import *
```

""" An exception used when we don't find a matching component in the source. """

```
NoSymbolMatch = "NoSymbolMatch"
```

```
class Src:
```

    """ The Src class is the specification of the source. It must contain two lists of two-tupples; `srctok`, which specifies the tokens to be recognized, and `srccomp`, which specifies the language components to be recognized (language components are made from tokens in srctok). It also contains an option dictionary.

    The `ws` token must be defined in `srctok`. We also have six tokens that are not defined in `srctok`; `bod` (beginning of document), `eod` (end of document), `bof` (beginning of file), `eof` (end of file), `bol` (beginning of line) and `eol` (end of line). These are automatically inserted in the input stream. """

```
""" The option dictionary for Src (with default values). """
option = {
    "startline": "1",        # Usual starts at first line
    "stopline": "-1",        # and stops at last line (-1).
    "tabsize": "8",          # Deafault tabular size is 8.
    "currcomp": ""}

""" Used in the token definitions. """
whitespaces = "[\011\013\014 ]+"
symbolstr = "[-:=;\.<>\+\*/!%,|&^~]"
symbols = "(\|)\|\[\|\]\|'\|\"\|`\|{\|}\|#"
name = "[a-zA-Z_][a-zA-Z0-9_]*"
escape = "\\\\\(" + symbols + "\|" + symbolstr + "\|[a-zA-Z0-9]+\)"
numbers = "0x[" + string.hexdigits + "]+"
numbers = numbers + "\|[0-9]*\.?[0-9]+L?\(e-?[0-9]+\)?"
keywords = "access\|and\|break\|class\|continue\|def\|del\|elif"
keywords = keywords + "\|else\|except\|exec\|finally\|for\|from"
keywords = keywords + "\|global\|if\|import\|in\|is\|lambda\|not"
keywords = keywords + "\|or\|pass\|print\|raise\|return\|try\|while"

""" Tokens recognized in the source language. """
srctok = [
    ("esc", (escape , [])),
    ("name",
     (name, [
         ("kw",
          (keywords, [
              ("def", ("def", [])),
              ("cls", ("class", [])),
              ("from", ("from", [])),
              ("import", ("import", []))])),
         ("r", ("r\|R", []))])),
    ("num", (numbers, [])),
    ("ws", (whitespaces, [])),
    ("symstr",
     (symbolstr, [
         ("period", ("\.", [])),
         ("comma", (",", [])),
         ("asterix", ("\*", []))])),
    ("sym",
     (symbols, [
         ("quote", ("'", [])),
         ("dquote", ('"', [])),
         ("bquote", ("`", [])),
         ("comchar", ("#", []))]))]

""" Source components recognized. We are using list and not dictionary because the order is important
(we search for components in this order). """
srccomp = [
    ("docstring", [
        ("#", "bol"),
        ("?", "ws"),
        ("%", [
            ("?", "r"),
            ("", "quote"),
            ("", "quote"),
            ("", "quote")]),
        ("*%", [
```

```
            ("^", [
                ("", "quote"),
                ("", "quote"),
                ("", "quote")])]),
        ("%", [
            ("", "quote"),
            ("", "quote"),
            ("", "quote")]),
        ("%", [
            ("?", "ws"),
            ("", "eol")])]),
    ("docstring", [
        ("#", "bol"),
        ("?", "ws"),
        ("%", [
            ("?", "r"),
            ("", "dquote"),
            ("", "dquote"),
            ("", "dquote")]),
        ("*%", [
            ("^", [
                ("", "dquote"),
                ("", "dquote"),
                ("", "dquote")])]),
        ("%", [
            ("", "dquote"),
            ("", "dquote"),
            ("", "dquote")]),
        ("%", [
            ("*", "ws"),
            ("", "eol")])]),
    ("quotepar", [
        ("", [
            ("?", "r"),
            ("", "quote"),
            ("", "quote"),
            ("", "quote")]),
        ("*", [
            ("^", [
                ("", "quote"),
                ("", "quote"),
                ("", "quote")])]),
        ("", [
            ("", "quote"),
            ("", "quote"),
            ("", "quote")])]),
    ("quotepar", [
        ("", [
            ("?", "r"),
            ("", "dquote"),
            ("", "dquote"),
            ("", "dquote")]),
        ("*", [
            ("^", [
                ("", "dquote"),
                ("", "dquote"),
                ("", "dquote")])]),
        ("", [
```

```
            ("", "dquote"),
            ("", "dquote"),
            ("", "dquote")])]),
    ("quotepar", [
        ("", [
            ("?", "r"),
            ("", "bquote"),
            ("", "bquote"),
            ("", "bquote")]),
        ("*", [
            ("^", [
                ("", "bquote"),
                ("", "bquote"),
                ("", "bquote")])]),
        ("", [
            ("", "bquote"),
            ("", "bquote"),
            ("", "bquote")])]),
    ("quotestring", [
        ("?", "r"),
        ("", "quote"),
        ("*", [
            ("^", "quote")]),
        ("", "quote")]),
    ("quotestring", [
        ("?", "r"),
        ("", "dquote"),
        ("*", [
            ("^", "dquote")]),
        ("", "dquote")]),
    ("quotestring", [
        ("?", "r"),
        ("", "bquote"),
        ("*", [
            ("^", "bquote")]),
        ("", "bquote")]),
    ("bol", [
        ("", "bol")]),
    ("eol", [
        ("", "eol")]),
    ("keyword", [
        ("", "kw")]),
    ("comment", [
        ("", "comchar"),
        ("*", [
            ("^", "eol")])]),
    ("function", [
        ("", "def"),
        ("", "ws"),
        ("", "name")]),
    ("class", [
        ("", "cls"),
        ("", "ws"),
        ("", "name")]),
    ("importstm", [
        ("", "import"),
        ("", "ws"),
        ("", [
```

```
            ("", "name"),
            ("*", [
                ("", "period"),
                ("", "name")]),
            ("*", [
                ("?", "ws"),
                ("", "comma"),
                ("?", "ws"),
                ("", [
                    ("", "name"),
                    ("*", [
                        ("", "period"),
                        ("", "name")])])])]),
    ("fromstm", [
        ("", "from"),
        ("", "ws"),
        ("", [
            ("", "name"),
            ("*", [
                ("", "period"),
                ("", "name")])]),
        ("", "ws"),
        ("", "import"),
        ("", "ws"),
        ("/", [
            ("", "asterix"),
            ("", [
                ("", "name"),
                ("*", [
                    ("?", "ws"),
                    ("", "comma"),
                    ("?", "ws"),
                    ("", "name")])])])])]

class Doc:
    """ The Doc class contains information about how to generate the document format. It must contain
    three different dictionaries; charmap to map illegal document characters to strings, tokmap to map tokens
    to document strings, and compmap to map language components to document text. There is also an option
    dictionary, used for example in the mapping of tokens (see mapTok()). """

    """ The option dictionary for Doc (with default values). """
    option = {
        "document": "1",          # Generate document by default
        "cls": "aasrc",           # using the "aasrc" class
        "clsopt": "",             # with no options
        "preamble": "",           # and no preamble
        "title": ""}              # using the default title.

    """ The character mapping to avoid illegal document (LaTeX) characters. """
    charmap = {
        "%": "\\%",
        "$": "\\$",
        "&": "\\&",
        "#": "\\#",
        "_": "{\\aaus}",
        "{": "{\\aalbrace}",
        "}": "{\\aarbrace}",
        "\\": "{\\aabs}",
        "^": "{\\aahat}",
```

```
        "~": "{\\aatilde}",
        "|": "{\\aabar}",
        "<": "{\\aalt}",
        ">": "{\\aagt}",
        "*": "{\\aast}"}

    """ The mapping from source tokens to document dependent tokens. The substrings (except token) is
    fetched from the option dictionary in this class. The first element in the tupple is the new name of the
    token (after the map). """
    tokmap = {
        "bod": ("", ("\\documentclass%(clsopt)s{%(cls)s}\n%(preamble)s" +
                     "\\begin{document}\n\\title{%(title)s}\n%(token)s")),
        "eod": ("", "%(token)s\\end{document}\n"),
        "bof": ("", "\\begin{aasrc}\n%(token)s"),
        "eof": ("", "%(token)s\\end{aasrc}\n"),
        "bol": ("bol", "%(token)s"),
        "eol": ("eol", "%(token)s"),
        "ws": ("ws", "\\aaws{%(token)s}")}

    """ The mapping from language to document components. Be aware that there should be a match
    between the number of substrings (the document format presented here and the the number of tuppels in
    the description list for the coresponding source language component (Src.srccomp). """
    compmap = {
        "docstring": "\\aadoc{%s}{%s}{%s}%s\\endaadoc{%s}%s",
        "quotepar": "\\aaqt{%s%s%s}",
        "quotestring": "\\aaqt{%s%s%s%s}",
        "bol": "\\aaline{%s}{",
        "eol": "}%s",
        "keyword": "\\aakw{%s}",
        "comment": "%s\\aacom{%s}",
        "function": "\\aakw{%s}%s\\aafunc{%s}",
        "class": "\\aakw{%s}%s\\aacls{%s}",
        "importstm": "\\aakw{%s}%s\\aamod{%s}",
        "fromstm": "\\aakw{%s}%s\\aamod{%s}%s\\aakw{%s}%s\\aanm{%s}"}

class SrcTok:
    """ Generate tokens from the source. """

    def __init__(self, input=None, src=None, doc=None):
        """ Initialize the SrcTok class. """
        if input:
            self.input = input
        else:
            self.input = sys.stdin
        if src:
            self.src = src
        else:
            self.src = Src()
        if doc:
            self.doc = doc
        else:
            self.doc = Doc()
        self.compiledtokens = self.compileSrcTokens(self.src.srctok)
        self.compiledall = regex.compile(string.join(map(
            lambda tokenit: tokenit[1][0],
            self.src.srctok), "\|"))
        self.linenum = 1
        while self.linenum < string.atoi(self.src.option["startline"]):
            if not self.input.readline(): break
```

```
        self.linenum = self.linenum + 1

    def compileSrcTokens(self, srctok):
        """ Make a compiled srctokens tree. """
        compiled = []
        for (name, spec) in srctok:
            compiled.append((name, regex.compile(spec[0]),
                             self.compileSrcTokens(spec[1])))
        return compiled

    def searchToken(self, line, pos):
        """ Search for next token. """
        return self.compiledall.search(line, pos)

    def bestToken(self, line, pos, compiled=[]):
        """ Find the best token match (most specialized) at this position. """
        if not compiled: compiled = self.compiledtokens
        for token in compiled:
            length = token[1].match(line, pos)
            if length > 0:
                if token[2]:
                    (nlength, nname) = self.bestToken(line, pos, token[2])
                    if nlength == length: return (nlength, nname)
                return (length, token[0])
        return (0, "")

    def decompLine(self, tokens, line):
        """ Decompose a line to a list of tokens. """
        tabsize = string.atoi(self.src.option["tabsize"])
        linepos = index = 0
        pos = self.searchToken(line, index)
        while pos != -1:
            (length, name) = self.bestToken(line, pos)
            if pos > index:
                tokens.append(("", line[index:pos]))
            if name == "ws":
                num = 0
                for wsi in range(pos, pos + length):
                    if line[wsi] == "\t":
                        num = num + (tabsize - ((linepos + num) % tabsize))
                    else:
                        num = num + 1
                linepos = linepos + num
                token = `num`
            else:
                linepos = linepos + length
                token = line[pos:pos+length]
            tokens.append((name, token))
            index = pos + length
            pos = self.searchToken(line, index)
        if index < len(line):
            tokens.append(("", line[index:len(line)]))

    def fetchTokens(self, tokens):
        """ Fetch a new line from the source and decompose it to tokens. Returns false if there ain't no line
        to fetch. """
        if (string.atoi(self.src.option["stopline"]) == -1 or
            self.linenum <= string.atoi(self.src.option["stopline"])):
            line = self.input.readline()
```

```
            debug.write("%s." % `self.linenum`)                                    473
            self.linenum = self.linenum + 1                                        474
        else:                                                                      475
            line = ""                                                              476
        if not line: return 0                                                      477
        tokens.append(("bol", `self.linenum - 1`))                                 478
        self.decompLine(tokens, line[:-1])                                         479
        tokens.append(("eol", "\n"))                                               480
        return 1                                                                   481
                                                                                   482
class Tok:                                                                         483
    """ A class to manage tokens (with possible check points). """                 484
                                                                                   487
    def __init__(self, srctok=None, pre=[], post=[]):                              488
        """ Initialize the Tok class. """                                          489
        if srctok:                                                                 492
            self.srctok = srctok                                                   493
        else:                                                                      494
            self.srctok = SrcTok()                                                 495
        self.src = self.srctok.src                                                 496
        self.doc = self.srctok.doc                                                 497
        self.tokenpos = -1                                                         498
        self.nextlist = []                                                         499
        if pre:                                                                    500
            self.tokens = pre                                                      501
        else:                                                                      502
            if self.doc.option["document"] != "0":                                 503
                self.tokens = [("bod", "")]                                        504
                debug.write("Generating document\n")                               505
            else:                                                                  506
                debug.write("Generating environment\n")                            507
                self.tokens = [("bof", "")]                                        508
        if post:                                                                   509
            self.posttok = post                                                    510
        else:                                                                      511
            if self.doc.option["document"] != "0":                                 512
                self.posttok = [("eod", "")]                                       513
            else:                                                                  514
                self.posttok = [("eof", "")]                                       515
                                                                                   516
                                                                                   517
    def checkPoint(self):                                                          518
        """ Make a check point. We must save the position off current token so we can rollback. """ 519
        self.nextlist.append(self.tokenpos)                                        523
                                                                                   524
    def commit(self):                                                              525
        """ Ok, we committed the sequence of tokens from last check point. """     526
        del self.nextlist[-1]                                                      529
                                                                                   530
    def rollBack(self):                                                            531
        """ Don't commit the token sequence. Rollback to the last check point. """ 532
        self.tokenpos = self.nextlist[-1]                                          535
        del self.nextlist[-1]                                                      536
                                                                                   537
    def next(self):                                                                538
        """ Fetch the next token. We may have to fetch a new line from the source file (with fetchTokens). 539
        """                                                                        540
        if self.tokenpos + 1 < len(self.tokens):                                   543
            self.tokenpos = self.tokenpos + 1                                      544
```

```
        else:                                                                   545
            if self.nextlist:                                                   546
                self.tokenpos = self.tokenpos + 1                               547
            else:                                                               548
                self.tokens = []                                                549
                self.tokenpos = 0                                               550
            if not self.srctok.fetchTokens(self.tokens):                        551
                if self.posttok:                                                552
                    self.tokens = self.tokens + self.posttok                    553
                    self.posttok = []                                           554
                else:                                                           555
                    raise IndexError                                            556
                                                                                557
    def current(self):                                                          558
        """ Returns the current token. """                                      559
        return self.tokens[self.tokenpos]                                       562
                                                                                563
class TokComp:                                                                  564
    """ Find components. """                                                    565
                                                                                568
    def __init__(self, tok=None):                                               569
        """ Initialize the TokComp class. """                                   570
        if tok:                                                                 573
            self.tok = tok                                                      574
        else:                                                                   575
            self.tok = Tok()                                                    576
        self.src = self.tok.src                                                 577
        self.doc = self.tok.doc                                                 578
                                                                                579
    def mapChars(self, text, start=0, stop=0):                                  580
        """ Using charmap to map illegal document cahracters to commands. """   581
        if stop == 0: stop = len(text)                                          584
        ttext = ""                                                              585
        for index in range(start, stop):                                        586
            try:                                                                587
                ttext = ttext + self.doc.charmap[text[index]]                   588
            except KeyError:                                                    589
                ttext = ttext + text[index]                                     590
        return ttext                                                            591
                                                                                592
    def mapTok(self, mod):                                                      593
        """ Using tokmap to map language tokens to document tokens. """         594
        (name, text) = self.tok.current()                                       597
        currcomp = self.src.option["currcomp"]                                  598
        if name == "bol" and not currcomp == "bol":                             599
            try:                                                                600
                tt = string.split(self.doc.compmap[currcomp], "%s")[0]          601
            except KeyError:                                                    602
                tt = ""                                                         603
            ttext = self.doc.compmap[name] % (text,) + tt                       604
            if "%" in mod or "/" in mod:                                        605
                ttext = ""              # Discard line number                   606
            elif "#" in mod:                                                    607
                ttext = text                                                    608
            return ttext                                                        609
        if name == "eol" and not currcomp == "eol":                             610
            try:                                                                611
                tt = string.split(self.doc.compmap[currcomp], "%s")[-1]         612
```

```
            except KeyError:
                tt = ""
            ttext = tt + self.doc.compmap[name] % (text,)
            if "%" in mod or "/" in mod:
                ttext = text
            return ttext
        if "#" in mod or "%" in mod:
            if name == "ws":
                text = " " * string.atoi(text)
            return text
        if name == "ws" and "/" in mod:
            text = " " * string.atoi(text)
            return text
        try:
            self.doc.option["token"] = self.mapChars(text)
            return self.doc.tokmap[name][1] % self.doc.option
        except KeyError:
            return self.mapChars(text)

    def matchComp(self, comp):
        """ Find one component matching tokens. The modifier specifies the type of the match (not, zero
        or more, single). Be aware that some modifiers are only interpreted by matcCompList. """
        self.tok.checkPoint()
        self.tok.next()
        text = ""
        try:
            if "^" in comp[0]:
                if self.tok.current()[0] != comp[1]:
                    text = self.mapTok(comp[0])
                else:
                    raise NoSymbolMatch
            elif "*" in comp[0]:
                self.tok.rollBack()      # We can match * with zero tokens
                self.tok.checkPoint()
                try:
                    while 1:
                        self.tok.checkPoint()
                        self.tok.next()
                        if self.tok.current()[0] != comp[1]:
                            self.tok.rollBack()
                            break
                        else:
                            self.tok.commit()
                            text = text + self.mapTok(comp[0])
                except IndexError:
                    pass
            else:
                if self.tok.current()[0] == comp[1]:
                    text = self.mapTok(comp[0])
                else:
                    raise NoSymbolMatch
        except NoSymbolMatch:
            self.tok.rollBack()
            raise NoSymbolMatch
        else:
            self.tok.commit()
            return text
```

```
    def matchCompList(self, comp):
        """ Find a list component matching tokens. The modifier specifies the type of the match (not, zero
        or more, single, ...). """
        self.tok.checkPoint()
        text = ""
        try:
            # ? and + are the same for both lists and strings
            if "?" in comp[0]:
                try:
                    text = self.matchCompList((comp[0][1:], comp[1]))
                except NoSymbolMatch:
                    pass
            elif "+" in comp[0]:
                try:
                    text = self.matchCompList((comp[0][1:], comp[1]))
                except NoSymbolMatch:
                    raise NoSymbolMatch
                else:
                    try:
                        ttext = self.matchCompList(
                            ("*" + comp[0][1:], comp[1]))
                    except NoSymbolMatch:
                        pass
                    else:
                        text = text + ttext
            # Use matchComp if the component is not a list
            elif type(comp[1]) is StringType:
                try:
                    text = self.matchComp(comp)
                except NoSymbolMatch:
                    raise NoSymbolMatch
            # Ok, we know it is a list
            elif "|" in comp[0]:
                for (mod, ccomp) in comp[1]:
                    try:
                        text = self.matchCompList((mod + comp[0][1:], ccomp))
                    except NoSymbolMatch:
                        continue
                    else:
                        break
                else:
                    raise NoSymbolMatch
            elif "^" in comp[0]:
                self.tok.checkPoint()
                for (mod, ccomp) in comp[1]:
                    try:
                        ttext = self.matchCompList((mod + comp[0][1:], ccomp))
                    except NoSymbolMatch:
                        self.tok.next()
                        text = text + self.mapTok(comp[0])
                        self.tok.commit()
                        break
                    else:
                        text = text + ttext
                else:
                    self.tok.rollBack()
                    raise NoSymbolMatch
            elif "*" in comp[0]:
```

```
                cont = 1
                while cont:
                    cont = 0; ttext = ""
                    self.tok.checkPoint()
                    for (mod, ccomp) in comp[1]:
                        try:
                            tttext = self.matchCompList(
                                (mod + comp[0][1:], ccomp))
                        except NoSymbolMatch:
                            self.tok.rollBack()
                            break
                        else:
                            ttext = ttext + tttext
                    else:
                        self.tok.commit()
                        text = text + ttext
                        cont = 1
            else:
                if comp[0]:
                    mmod = comp[0]
                else:
                    mmod = ""
                for (mod, ccomp) in comp[1]:
                    try:
                        ttext = self.matchCompList((mod + mmod, ccomp))
                    except NoSymbolMatch:
                        raise NoSymbolMatch
                    else:
                        text = text + ttext
        except NoSymbolMatch:
            self.tok.rollBack()
            raise NoSymbolMatch
        else:
            self.tok.commit()
            return text

class Comp:
    """ Generates components (a name and a texttupple) from tokens. """

    def __init__(self, tokcomp=None):
        """ Initialize the TokComp class. """
        if tokcomp:
            self.tokcomp = tokcomp
        else:
            self.tokcomp = TokComp()
        self.tok = self.tokcomp.tok
        self.src = self.tokcomp.src
        self.doc = self.tokcomp.doc
        self.ftokcomp = {}
        for index in range(len(self.src.srccomp)):
            tlist = self.src.srccomp[index][1]
            try:
                self.appendFtok(index, tlist)
            except KeyError:
                self.ftokcomp = {}
                break
        if self.ftokcomp:
            debug.write("Using speedup\n")
```

```
796
797 def appendFtok(self, index, tlist):
798     if tlist[0][0] != "":
799         raise KeyError
800     if type(tlist[0][1]) is StringType:
801         try:
802             self.ftokcomp[tlist[0][1]].append(index)
803         except KeyError:
804             self.ftokcomp[tlist[0][1]] = [index]
805     else:
806         self.appendFtok(index, tlist[0][1])
807
808 def fetchComp(self, comp):
809     """ fetch a component by name. """
812     self.tok.checkPoint()
813     texttupple = ()
814     try:
815         for (mod, ccomp) in comp:
816             try:
817                 text = self.tokcomp.matchCompList((mod, ccomp))
818             except NoSymbolMatch:
819                 self.tok.rollBack()
820                 raise NoSymbolMatch
821             else:
822                 texttupple = texttupple + (text,)
823         else:
824             self.tok.commit()
825             return texttupple
826     except (KeyError, NoSymbolMatch):
827         raise NoSymbolMatch
828
829 def searchComp(self):
830     """ Search for components from in src.srccomp. Use speedup if available (based on value of first
        token). """
834     comprange = []
835     if self.ftokcomp:
836         self.tok.checkPoint()
837         self.tok.next()
838         try:
839             comprange = self.ftokcomp[self.tok.current()[0]]
840         except KeyError:
841             pass
842         self.tok.rollBack()
843     if not comprange:
844         comprange = range(len(self.src.srccomp))
845     for index in comprange:
846         (name, comp) = self.src.srccomp[index]
847         self.src.option["currcomp"] = name
848         try:
849             texttupple = self.fetchComp(comp)
850         except NoSymbolMatch:
851             continue
852         else:
853             return (name, texttupple)
854     self.tok.next()
855     self.src.option["currcomp"] = ""
856     return ("", (self.tokcomp.mapTok(""),))
857
```

```
class CompDoc:                                                          858
    """ Map characters, tokens and components to document format. """   859
                                                                        862
    def __init__(self, output=None, comp=None):                        863
        """ Initialize the CompDoc class. """                          864
        if output:                                                      867
            self.output = output                                        868
        else:                                                           869
            self.output = sys.stdout                                    870
        if comp:                                                        871
            self.comp = comp                                            872
        else:                                                           873
            self.comp = Comp()                                          874
        self.doc = self.comp.doc                                        875
                                                                        876
    def mapComp(self, name, texttupple):                                877
        """ Using compmap to map language components to document components. """ 878
        try:                                                            881
            return self.doc.compmap[name] % texttupple                  882
        except KeyError:                                                883
            return string.joinfields(texttupple, "")                    884
                                                                        885
                                                                        886
    def printDoc(self):                                                 887
        debug.write("Starting\n")                                       888
        while 1:                                                        889
            try:                                                        890
                (name, texttupple) = self.comp.searchComp()             891
                sys.stdout.write(self.mapComp(name, texttupple))        892
            except IndexError:                                          893
                break                                                   894
        debug.write("\nDone\n")                                         895
                                                                        896
class Debug:                                                            897
    """ Handy stuffs """                                               898
                                                                        901
    def __init__(self):                                                 902
        """ Initialize the Debug class. """                            903
        self.write = sys.stderr.write                                   906
                                                                        907
# Make an instance of Debug                                             908
debug = Debug()                                                         909
                                                                        910
                                                                        911
""" If this is the main file, we do it with default values. """        912
if __name__ == "__main__":                                              915
    CompDoc().printDoc()                                                916
```